Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
La Batalla Naval Individual

2 de diciembre de 2024

Damaris Juares
108566

Lucas Perez Esnaola
107990

Índice

1. Introducción

1.1. Contexto

Los hermanos siguieron creciendo. Mateo también aprendió sobre programación dinámica, y cada uno aplicaba la lógica sabiendo que el otro también lo hacía. El juego de las monedas se tornó aburrido en cuánto notaron que siempre ganaba quien empezara, o según la suerte. Los años pasaron, llegó la adolescencia y empezaron a tener gustos diferentes. En general, jugaban a juegos individuales. En particular, Sophia estaba muy enganchada con un juego inventado en Argentina por Jaime Poniachik (uno de los fundadores de Ediciones de Mente) en 1982: La Batalla Naval Individual.

En dicho juego, tenemos un tablero de nxm casilleros, y k barcos. Cada barco i tiene b_i de largo. Es decir, requiere de b_i casilleros para ser ubicado. Todos los barcos tienen 1 casillero de ancho. El tablero a su vez tiene un requisito de consumo tanto en sus filas como en sus columnas. Si en una fila indica un 3, significa que deben haber 3 casilleros de dicha fila siendo ocupados. Ni más, ni menos. No podemos poner dos barcos de forma adyacente (es decir, no pueden estar contiguos ni por fila, ni por columna, ni en diagonal directamente). Debemos ubicar todos los barcos de tal manera que se cumplan todos los requisitos. A continuación mostramos un ejemplo de un juego resuelto:

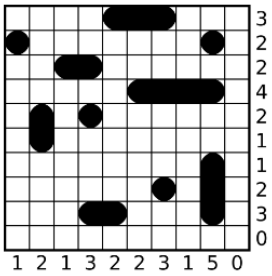

Figura 1: Ejemplo de Batalla Naval Individual resuelta.

1.2. Consigna

Para los primeros dos puntos considerar la versión de decisión del problema de La Batalla Naval: Dado un tablero de nxm casilleros, y una lista de k barcos (donde el barco i tiene b_i de largo), una lista de restricciones para las filas (donde la restricción j corresponde a la cantidad de casilleros a ser ocupados en la fila j) y una lista de restricciones para las columnas (símil filas, pero para columnas), ¿es posible definir una ubicación de dichos barcos de tal forma que se cumplan con las demandas de cada fila y columna, y las restricciones de ubicación?

1. Demostrar que el Problema de la Batalla Naval se encuentra en NP.
2. Demostrar que el Problema de la Batalla Naval es, en efecto, un problema NP-Completo. Si se hace una reducción involucrando un problema no visto en clase, agregar una (al menos resumida) demostración que dicho problema es NP-Completo.
3. Escribir un algoritmo que, por backtracking, obtenga la solución óptima al problema (valga la redundancia) en la versión de optimización: Dado un tablero de nxm casilleros, y una lista de k barcos (donde el barco i tiene b_i de largo) una lista de las demandas de las n filas y una lista de las m demandas de las columnas, dar la asignación de posiciones de los barcos de tal forma que se reduzca al mínimo la cantidad de demanda incumplida. Pueden no utilizarse todos los barcos. Si simplemente no se cumple que una columna que debería tene 3 casilleros ocupados tiene 1, entonces contará como 2 de demanda incumplida. Por el contrario, no

está permitido exceder la cantidad demandada. Generar sets de datos para corroborar su correctitud, así como tomar mediciones de tiempos.

4. Escribir un modelo de programación lineal que resuelva el problema de forma óptima. Ejecutarlo para los mismos sets de datos para corroborar su correctitud. Tomar mediciones de tiempos y compararlas con las del algoritmo que implementa Backtracking.

5. John Jellicoe (almirante de la Royal Navy durante la batalla de Jutlandia) nos propone el siguiente algoritmo de aproximación: Ir a fila/columna de mayor demanda, y ubicar el barco de mayor longitud en dicha fila/columna en algún lugar válido. Si el barco de mayor longitud es más largo que dicha demanda, simplemente saltearlo y seguir con el siguiente. Volver a aplicar hasta que no queden más barcos o no haya más demandas a cumplir.

   Este algoritmo sirve como una aproximación para resolver el problema de La Batalla Naval. Implementar dicho algoritmo, analizar su complejidad y analizar cuán buena aproximación es. Para esto, considerar lo siguiente: Sea I una instancia cualquiera del problema de La Batalla Naval, y $z(I)$ una solución óptima para dicha instancia, y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leqslant r(A)$ para todas las instancias posibles. Calcular $r(A)$ para el algoritmo dado, demostrando que la cota está bien calculada. Realizar mediciones utilizando el algoritmo exacto y la aproximación, con el objetivo de verificar dicha relación. Realizar también mediciones que contemplen volúmenes de datos ya inmanejables para el algoritmo exacto, a fin de corroborar empíricamente la cota calculada anteriormente.

6. Opcional: Implementar alguna otra aproximación (o algoritmo greedy) que les parezca de interés. Comparar sus resultados con los dados por la aproximación del punto anterior. Indicar y justificar su complejidad. No es obligatorio hacer este punto para aprobar el trabajo práctico (pero si resta puntos no hacerlo).

7. Agregar cualquier conclusión que parezca relevante.

2. Demostracion NP

Para demostrar que la Batalla Naval Individual se encuentra en NP, debemos demostrar que existe un certificador eficiente, es decir, que debemos poder validar en tiempo polinomial que una solución dada cumple con las restricciones del problema.

Estas restricciones son:

- No existen barcos ubicados de forma diagonal.
- Se cumple con las demandas de fila.
- Se cumple con las demandas de columna
- Todos los barcos pedidos estan ubicados en el tablero

Además, representamos a los barcos en el tablero como una serie de 1s contiguos (tanto verticales como horizontales), y al agua en el tablero con 0.

2.1. Algoritmo certificador eficiente

A continuación se muestra el código del certificador eficiente planteado. El certificador recibe:

- El tablero de $n \times m$ casilleros
- La lista de k barcos, donde el barco i tiene b_i de largo
- La lista de restricciones para las filas, donde la restriccion j corresponde a la cantidad de casilleros a ser ocupados por la fila j
- La lista de restricciones para las columnas, con el mismo formato que para las filas.

```
def certificador_eficiente(tablero, barcos, requisitos_fil, requisitos_col):

    # Chequeo barcos diagonales -> O(nxm)
    if chequeo_barcos_diagonales(tablero) == False:
        return False

    # Chequeo requisitos de fila -> O(nxm)
    for i in range(len(tablero)):
        if tablero[i].count(1) != requisitos_fil[i]:
            return False

    # Chequeo requisitos de columna -> O(nxm)
    for j in range(len(tablero[0])):
        sum_j = sum(fila[j] for fila in tablero)
        if (sum_j != requisitos_col[j]):
            return False

    # Chequeo barcos asignados correctamente -> O(nxm)
    if chequear_barcos(tablero, barcos) == False:
        return False

    return True
```

2.2. Detalles del algoritmo

2.2.1. Chequeo barcos diagonales

La siguiente función verifica que no haya barcos ubicados de forma diagonal en el tablero:

```python
def chequeo_barcos_diagonales(tablero):
    n = len(tablero)
    m = len(tablero[0])

    for i in range(n):
        for j in range(m):

            if tablero[i][j] == 1:

                # Superior izq -> O(1)
                if i > 0 and j > 0 and tablero[i-1][j-1] == 1:
                    return False

                # Superior der -> O(1)
                if i > 0 and j < m-1 and tablero[i-1][j+1] == 1:
                    return False

                # Inferior izq -> O(1)
                if i < n-1 and j > 0 and tablero[i+1][j-1] == 1:
                    return False

                if i < n-1 and j < m-1 and tablero[i+1][j+1] == 1: -> O(1)
                    return False

    return True
```

Al iterar sobre todas las posiciones del tablero, la complejidad del algoritmo es de $\mathcal{O}(n \times m)$

2.2.2. Chequeo requisitos de fila

El siguinte fragmento de código verifica que el tablero cumpla con las demandas de fila pasadas como argumento.

```python
    # Chequeo requisitos de fila -> O(nxm)
    for i in range(len(tablero)):
        if tablero[i].count(1) != requisitos_fil[i]:
            return False
```

Al iterar sobre todas las filas del tablero, y usar la función *count*() (que tiene una complejidad de $\mathcal{O}(m)$, la verificacion tiene una complejidad de $\mathcal{O}(n \times m)$

2.2.3. Chequeo requisitos de columna

El siguinte fragmento de código verifica que el tablero cumpla con las demandas de columna pasadas como argumento.

```python
    # Chequeo requisitos de columna -> O(nxm)
    for j in range(len(tablero[0])):
        sum_j = sum(fila[j] for fila in tablero)
        if (sum_j != requisitos_col[j]): # O(1)
            return False
```

Primero itera sobre las columnas del tablero en $\mathcal{O}(m)$ y luego suma los elementos de cada columna iterando sobre las filas del tablero, dando una complejidad total de $\mathcal{O}(n \times m)$

2.2.4. Chequeo de barcos

La siguiente función verifica que todos los barcos de la lista pasada como argumento esten ubicados en el tablero:

```python
def chequear_barcos(tablero, barcos):
    n = len(tablero)
    m = len(tablero[0])

    for i in range(n):
```

```
        for j in range(m):

            if len(barcos) == 0:
                break

            if tablero[i][j] == 1:
                largo_barco = 1
                tablero[i][j] = 0
                # Barco horizontal
                if j < m-1 and tablero[i][j+1] == 1:
                    while j < m-1:
                        if tablero[i][j+1] == 1:
                            largo_barco += 1
                            tablero[i][j+1] = 0
                            j += 1
                        else:
                            break

                # Barco vertical
                elif i < n-1 and tablero[i+1][j] == 1:
                    while i < n-1:
                        if tablero[i+1][j] == 1:
                            largo_barco += 1
                            tablero[i+1][j] = 0
                            i += 1
                        else:
                            break

                if barcos.count(largo_barco) > 0:
                    barcos.remove(largo_barco) # O(n)
                else:
                    # No existe un barco con esa longitud
                    return False

    # Quedaron barcos sin asignar al tablero
    if len(barcos) != 0:
        return False

    return True
```

Por cada casillero ocupado, primero se analiza si es un barco vertical o horizontal, para luego buscar el tamaño del barco. Si en la lista de barcos existe un barco con ese tamaño, lo elimina de la lista, sino devuelve False. Al finalizar, si quedaron barcos sin remover de la lista, quiere decir que no todos los barcos fueron ubicados correctamente por lo que devuelve False, de lo contrario devuelve True.

Al iterar sobre todos los casilleros del tablero, la complejidad de la función es de $\mathcal{O}(n \times m)$

2.3. Conclusión

La complejidad del validador propuesto es $\mathcal{O}(n \times m)$, lo que es polinomial respecto de los valores de entrada, por lo que podemos afirmar que La Batalla Naval Individual está en NP.

3. Demostración NP Completo

A continuación, se demostrará que el problema de la Batalla Naval es un problema NP-Completo, utilizando el problema de Bin-Packing en su versión unaria, como fue recomendado.

Para empezar, primero se debe demostrar que el problema de la Batalla Naval está en NP. Esto fue demostrado en el punto anterior.

Luego, se debe demostrar que el problema de la Batalla Naval está en NP-Completo mediante una reducción de un problema NP-Completo a el problema de la Batalla Naval.

3.1. Problema de Bin-Packing en version unaria

Se eligió al problema de Bin-Packing en su version unaria para realizar la reducción al problema de la Batalla Naval. El problema se explica de la siguiente manera:

Dados un conjunto de números S, cada uno expresado en código unario, una cantidad de bins B, expresada en código unario, y la capacidad de cada bin C, expresada en código unario; donde la suma del conjunto de números es igual a C*B, debe decidir si puede cumplirse que los números pueden dividirse en B bins, subconjuntos disjuntos, tal que la suma de elementos de cada bin es exactamente igual a la capacidad C.

3.2. Bin-Packing $\leq_P$ Batalla Naval

Debemos demostrar que utilizando una 'caja negra' que resuelve el problema de la Batalla Naval, se puede resolver el problema de Bin-Packing en su versión unaria.

Para esto, debemos transformar polinomialmente la información brindada al problema de Bin-Packing hacia el problema de la Batalla Naval. Se plantea la siguiente transformación:

- Ubicar un barco en una fila del tablero equivale a asignar un numero s_i a un bin.
- El conjunto de números del problema de Bin-Packing serían equivalente al conjunto de barcos en el problema de la Batalla Naval. Es decir, si en una instancia de Bin-Packing tenemos el conjunto {11, 111, 11, 1}, en una instancia de la Batalla Naval tendríramos los barcos {2, 3, 2, 1}.
- Por cada bin, habría una fila en el tablero de la batala naval, y tambien habría una fila más debajo de esta fila. Por lo que la cantidad de bins B (pasada a decimal) sería equivalente a $n/2$, siendo n el numero de filas del tablero.
- La capacidad C de cada bin (pasada a decimal) sería equivalente a la demanda de las filas en el tablero asignadas a cada bin. Las filas intermedias entre bins tendrían 0 de demanda, para evitar barcos adyacentes verticalmente.
- Para evitar barcos adyacentes horizontalmente, la cantidad de columnas m del tablero será igual a $2C - 1$.
- Para las demandas de columnas, se sabe que al ubicar un barco en una celda este cumple 1 de demanda tanto para la fila como para la columna donde esta ubicado. Por lo tanto, el total de la demanda a cumplir por las columnas es $\sum S$. Se debe distribuir uniformemente la demanda a cada columna, con el cálculo de $\frac{\sum S}{2C-1}$ para la demanda de cada columna.
- Si se pueden ubicar los barcos en el tablero cumpliendo con la demanda, que resultan de la transformación planteada, entonces se pueden dividir el conjunto de numeros S en B bins tal que la suma de los elementos de cada bin es exactamente igual a C.

Las anteriores transformaciones se pueden realizar con operaciones polinomiales.

Para demostrar que la reducción es correcta, debemos demostrar que si para cualquier instancia del problema de la Batalla Naval que resulte de la reducción planteada, si existe una solución implica que para la instancia original del problema de Bin-Packing en su version unaria también existe una solución, y viceversa.

3.2.1. BN → BP

Si para la instancia del problema de la Batalla Naval que resulta de la reducción planteada existe una solución, esto implica que:

Debido a que cada fila representa un bin, y cada demanda de fila representa la capacidad de ese bin, si existe una solución quiere decir que todas la demandas de las filas fueron cumplidas, y por lo tanto, la suma del tamaño de los barcos ubicados en cada fila es igual a la demanda de cada fila. Esto se traduce a que la suma de los elementos ubicados en cada bin es igual a la capacidad C, y por lo tanto, existe una solución para el problema de Bin-Packing en su version unaria.

3.2.2. BP → BN

Si para una instancia arbitraria del problema de Bin-Packing en su version unaria existe solución, esto implica que:

- El subconjunto de numeros pudo dividirse en los B bines. Esto se traduce a que los barcos pudieron ser ubicados en las filas del tablero.
- La suma de los elementos de cada bin es exactamente igual a la capacidad C. Esto se traduce a que cada fila y columna tienen su demanda cumplida.
- Por la definición de la reducción no se infringen las restricciones de barcos adyacentes ni diagonales.

Y por lo tanto, existe una solución para el problema de la Batalla Naval para la instancia resultante de la reducción planteada.

3.3. Conclusión

Hemos demostrado que la reducción es correcta y por ende el problema de Bin-Packing en su version unaria puede ser reducido polinomialmente al problema de la Batalla Naval.

Por lo tanto, podemos afirmar que el problema de la Batalla Naval es un problema NP-Completo.

4. Backtracking

En la siguiente sección se resolverá la versión de optimización de la Batalla Naval Individual mediante un algoritmo de Backtracking. Este algoritmo explora todas las posibilidades de solucion, realizando podas para evitar calcular soluciones que ya no son válidas o mejores que la solucion actual, y devuelve la solución óptima.

4.1. Aclaraciones previas

Contrario a puntos anteriores, los barcos se representarán en el tablero con un número de indice. De esta manera se indica no sólo la posición de los barcos en el tablero, sino qué barco esta ubicado.

4.2. Detalles del algoritmo

A continuación se mostrará el código del algoritmo planteado:

```
def batalla_naval_bt(tablero, barcos, demanda_fila, demanda_columna, mejor_demanda_inc, demanda_inc, indice, mejor_tablero):
    if mejor_demanda_inc == 0:
        return mejor_tablero, mejor_demanda_inc

    if not barcos:
        demanda_inc = calcular_demanda_incumplida(demanda_fila, demanda_columna)
        if demanda_inc < mejor_demanda_inc:
            mejor_demanda_inc = demanda_inc
            mejor_tablero = copy.deepcopy(tablero)
        return mejor_tablero, mejor_demanda_inc

    barcos_filtrados = filtrar_barcos_por_demanda(barcos, demanda_fila, demanda_columna)
    if not barcos_filtrados:
        return mejor_tablero, mejor_demanda_inc

    # Si con los barcos que me quedan no puedo mejorar la mejor demanda incumplida, corto la rama
    if calcular_demanda_incumplida(demanda_fila, demanda_columna) - sum(barcos_filtrados)*2 >= mejor_demanda_inc:
        return mejor_tablero, mejor_demanda_inc

    posiciones = calcular_posibles_posiciones(tablero, len(tablero), len(tablero[0]), barcos_filtrados[0], demanda_fila, demanda_columna)

    mejor_tablero_actual = mejor_tablero

    for posicion in posiciones:

        if calcular_demanda_incumplida(demanda_fila, demanda_columna) - sum(barcos_filtrados)*2 >= mejor_demanda_inc:
            return mejor_tablero, mejor_demanda_inc

        nuevo_tablero = marcar_barco_en_tablero(tablero, indice, barcos_filtrados[0], posicion)

        nueva_demanda_fila, nueva_demanda_columna = actualizar_demandas(demanda_fila, demanda_columna, barcos_filtrados[0], posicion)

        # Si el barco actual no se puede ubicar, todos los barcos con igual tamano tampoco, obtengo el primero distinto
        if posicion == None:
            i_prox_barco =  obtener_indice_prox_barco(barcos_filtrados)
        else:
            i_prox_barco = 1

        nuevo_tablero, nuevo_demanda_inc = batalla_naval_bt(nuevo_tablero, barcos_filtrados[i_prox_barco:], nueva_demanda_fila, nueva_demanda_columna, mejor_demanda_inc, demanda_inc, indice+1, mejor_tablero_actual)
```

```
40
41          if nuevo_demanda_inc < mejor_demanda_inc:
42              mejor_demanda_inc = nuevo_demanda_inc
43              mejor_tablero = nuevo_tablero
44
45      return mejor_tablero, mejor_demanda_inc
```

El algoritmo funciona de la siguiente manera:

1. La condición de corte de la función recursiva es al no quedar barcos por ubicar, se calcula la demanda incumplida de la solución actual y si es menor a la mejor, se actualiza.
2. La primera poda se ve al filtrar los barcos por demanda. Si hay barcos cuyo tamaño es mayor a la máxima demanda permitida por filas o columnas, se eliminan de la solución.
3. La segunda poda se ve al cortar la rama de solución si es que con los barcos que me quedan por ubicar no puedo mejorar la demada incumplida por la mejor solución.
4. Luego, se calculan las posibles posiciones para el barco actual. Para calcular esto se tienen en cuenta los barcos ya ubicados en el tablero, las demandas de filas y columnas actuales de la solución, y las restricciones de barcos contiguos y diagonales.
5. Se itera sobre las posiciones calculadas, se marca el barco en el tablero en la posicion indicada y se actualizan las demandas.
6. Previo al llamado recursivo, se realiza otra poda. Si no hay posiciones válidas para ubicar el barco actual (es decir posicion = None), quiere decir que ningún barco del mismo tamaño va a poder ser ubicado. Para eso se calcula el índice del primer barco con tamaño distinto al actual y se llama a la función recursiva con ese barco.
7. Al volver del llamado recursivo, se analiza si la solución devuelta es mejor que la mejor solución actual, y se actualiza, para luego devolver la mejor solución.

4.3. Resultados

A continuación, se mostrarán los tiempos de ejecución del algoritmo de backtracking.

4.3.1. Generación de sets de datos

Para medir con mas precisión los tiempos de ejecución, se generaron sets de datos propios con la siguiente función:

```
1  # Genera n sets de datos de tamano incremental para el problema de la Batalla Naval
       .
2  # Si se especifica una semilla, se utilizara para generar los datos.
3  def generar_set_datos_batalla_naval(semilla, n):
4      if semilla != None:
5          seed(semilla)
6
7      set_datos = []
8
9      for i in range(1, n+1):
10         n = i * 3
11         m = i * 3
12         k = i * 2
13         barcos = [randint(1, min(i * 3, 16)) for _ in range(k)]
14         demandas_filas = [randint(0, min(i * 3, 15)) for _ in range(n)]
15         demandas_columnas = [randint(0, min(i * 3, 15)) for _ in range(m)]
16
17         set_datos.append((barcos, demandas_filas, demandas_columnas))
18
19     return set_datos
```

De esta forma, se generan sets de datos que van incrementando iterativamente el tamaño del tablero y la cantidad de barcos, aplicandole un límite al tamaño de barcos y de demandas para que no sean excesivas.

Utilizando el método mencionado anteriormente, se generaron 10 sets de datos como se muestran a continuación:

N° Set	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Número de Barcos	2	4	6	8	10	12	14	16	18	20
Tamaño del Tablero	3x3	6x6	9x9	12x12	15x15	18x18	21x21	24x24	27x27	30x30

Cuadro 1: Sets de datos generados.

4.3.2. Mediciones

Los resultados de las mediciones de tiempo obtenidas se muestran a continuación:

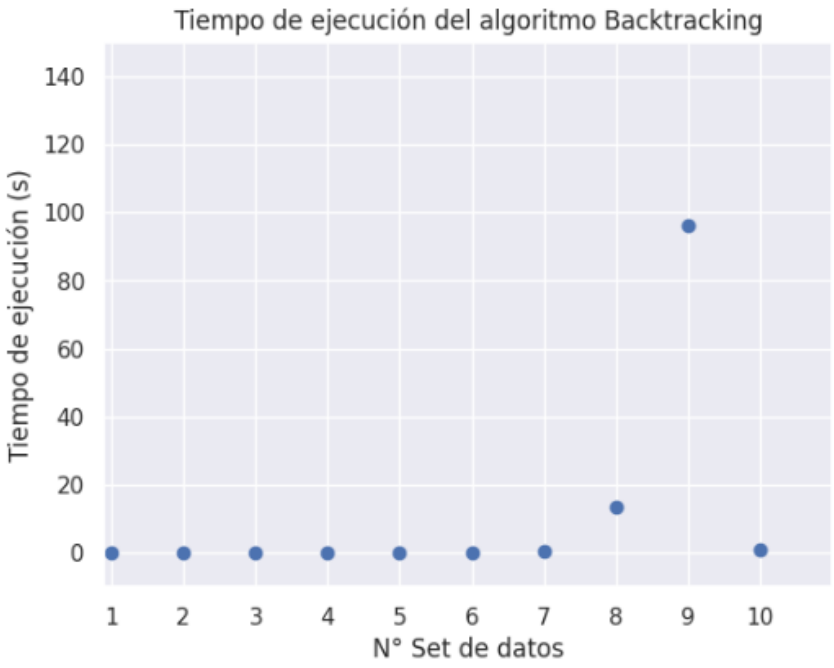

Figura 2: Mediciones en segundos del tiempo de ejecución del algoritmo Backtracking

4.4. Anánilis de los resultados

Analizando el gráfico mostrado de los tiempos de ejecución del algoritmo de backtracking, podemos ver claramente como el algoritmo funciona casi instantáneamente para sets de datos de tamaño pequeño, pero a medida que se usan sets de datos mas grandes, el algoritmo tarda considerablemente más. Esto demuestra la complejidad teórica exponencial de los algoritmos de backtracking.
También podemos mencionar que hay ciertas configuraciones de barcos y demandas que hacen que el algoritmo tarde menos, como vemos en el set de datos 10, que por más de ser de mayor tamaño que los sets de datos 8 y 9, el tiempo de ejecución mucho más rápido.

5. Programación Lineal

En la siguiente sección se resolverá la versión de optimización de la Batalla Naval Individual mediante un algoritmo de Programación Lineal Entera. Este algoritmo define una serie de ecuaciones lineales, que representan variables del problema y sus restricciones, y luego busca los valores de las variables definidas que optimizen una función objetivo. Para resolverlo se utilizó la librería de Python PULP.

5.1. Modelo planteado

5.1.1. Variables del modelo

Para resolver el problema de optimización de la Batalla Naval Individual usando Programación Lineal Entera, se definieron las siguientes variables:

- $x_{i,j,b,o}$: Indica si el barco $\boldsymbol{b}$ tiene como posición inicial al casillero $(\boldsymbol{i,j})$ del tablero y con orientación $\boldsymbol{o}$. La orientación representa si un barco esta ubicado vertical u horizontalmente.
- u_i: Representa la demanda incumplida de la fila i. Es un valor entero.
- v_j: Representa la demanda incumplida de la columna j. Es un valor entero.

Se definen las siguientes restricciones para las variables:

5.1.2. Restricciones del modelo

Por cada fila i del tablero, la suma entre las celdas ocupadas en esa fila y la demanda incumplida de la fila (u_i), debe ser igual a la demanda de la fila i:

$\sum_{j=0}^{m} \sum_{b=0}^{k} x_{i,j,b,0} * barcos[b] + \sum_{j=1}^{m} \sum_{b=0}^{k} \sum_{l=0}^{barcos[b]} x_{i-l,j,b,1} + u_i = demanda\_filas[i], \forall i$

Por cada columna j del tablero, la suma entre las celdas ocupadas en esa columna y la demanda incumplida de la columna (v_j), debe ser igual a la demanda de la columna j:

$\sum_{i=0}^{n} \sum_{b=0}^{k} x_{i,j,b,1} * barcos[b] + \sum_{i=0}^{n} \sum_{b=0}^{k} \sum_{l=0}^{barcos[b]} x_{i,j-l,b,0} + v_j = demanda\_columnas[j], \forall j$

Cada barco debe tener una única posicion y orientacion inicial:

$\sum_{i=0}^{n} \sum_{j=0}^{m} \sum_{o=0}^{1} x_{i,j,b,o} \leq 1, \quad \forall b$

No puede haber dos barcos con la misma posicion en el tablero:

$\sum_{b=0}^{k} \sum_{o=0}^{1} x_{i,j,b,o} \leq 1, \quad \forall i, j$

Ningun barco puede estar ubicado de forma contigua o diagonal con otro barco:

Caso horizontal $(o = 0)$:

$$\forall (i, j, b) \text{ con } j + \text{barcos}[b] \leq m : \sum_{\substack{(ni,nj,b',o') \\ b' \neq b}} x[ni, nj, b', o'] \leq 1 - x[i, j, b, 0],$$

donde:

$$(ni, nj) = (i + \Delta_i, j + l + \Delta_j), \quad \Delta_i, \Delta_j \in \{-1, 0, 1\}, \quad l \in \{0, \ldots, \text{barcos}[b] - 1\}.$$

Caso vertical $(o = 1)$:

$$\forall (i,j,b) \text{ con } i + \text{barcos}[b] \leq n : \sum_{\substack{(ni,nj,b',o') \\ b' \neq b}} x[ni,nj,b',o'] \leq 1 - x[i,j,b,1],$$

donde:

$$(ni,nj) = (i + l + \Delta_i, j + \Delta_j), \quad \Delta_i, \Delta_j \in \{-1,0,1\}, \quad l \in \{0,\ldots,\text{barcos}[b]-1\}.$$

Por último, como función objetivo se planteó la minimización de las demandas incumplidas: $\sum_{i=0}^{n} u_i + \sum_{j=0}^{m} v_j$

5.2. Detalles del algoritmo

A continuación se mostrará el código del algoritmo de programación lineal entera planteado:

```
def batalla_naval_individual_pl(n, m, k, barcos, demandas_filas, demandas_columnas):

    # Crear el problema
    problema = LpProblem("Minimizar_demanda_incumplida", LpMinimize)

    # Variables de decision
    # x -> Barco b en la posicion (i, j) y orientacion o
    # u -> Demanda incumplida en fila i
    # v -> Demanda incumplida en columna j
    x = LpVariable.dicts("x", ((i, j, b, o) for i in range(n) for j in range(m) for b in range(k) for o in [0, 1]
        if es_posicion_valida(barcos, i, j, b, o, demandas_filas, demandas_columnas, n, m)), cat=LpBinary)

    u = LpVariable.dicts("u", (i for i in range(n)), lowBound=0, cat=LpInteger)
    v = LpVariable.dicts("v", (j for j in range(m)), lowBound=0, cat=LpInteger)

    # Funcion objetivo: Minimizar demanda incumplida
    problema += (
        lpSum(u[i] for i in range(n)) +
        lpSum(v[j] for j in range(m))
    ), "Minimizar_Demanda_Incumplida"

    # Restricciones de demanda en filas
    for i in range(n):
        problema += (
            lpSum(
                x[i, j, b, 0] * barcos[b] for j in range(m) for b in range(k) if (i,j,b,0) in x if j + barcos[b] <= m
            ) +
            lpSum(
                x[i - l, j, b, 1] for j in range(m) for b in range(k) for l in range(barcos[b]) if (i-l,j,b,1) in x if 0 <= i - l < n
            ) +
            u[i] == demandas_filas[i]
        )

    # Restricciones de demanda en columnas
    for j in range(m):
        problema += (
            lpSum(
                x[i, j, b, 1] * barcos[b] for i in range(n) for b in range(k) if (i,j,b,1) in x if i + barcos[b] <= n
            ) +
            lpSum(
                x[i, j - l, b, 0] for i in range(n) for b in range(k) for l in range(barcos[b]) if (i,j-l,b,0) in x if 0 <= j - l < m
            ) +
```

```
            v[j] == demandas_columnas[j]
        )


    # Restricciones de unicidad de los barcos
    for b in range(k):
        problema += (
            lpSum(x[i, j, b, o] for i in range(n) for j in range(m) for o in [0, 1]
     if (i,j,b,o) in x) <= 1
        )

    # Restricciones de no solapamiento
    for i in range(n):
        for j in range(m):
            problema += (
                lpSum(
                    x[i, j, b, o] for b in range(k) for o in [0, 1] if (i,j,b,o) in
     x
                ) <= 1
            ), f"No_Solapamiento_{i}_{j}"

    # Restricciones para evitar barcos contiguos y diagonales
    for (i, j, b, o) in x:
        if o == 0 and j + barcos[b] <= m:  # Barco horizontal
            for l in range(barcos[b]):
                for di, dj in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1)
    , (1, 0), (1, 1)]:
                    ni, nj = i + di, j + l + dj
                    if 0 <= ni < n and 0 <= nj < m:
                        problema += (
                            lpSum(x[ni, nj, b2, o2] for (ni2, nj2, b2, o2) in x if
    ni2 == ni and nj2 == nj and b2 != b) <= 1 - x[i, j, b, o]
                        )
        elif o == 1 and i + barcos[b] <= n:  # Barco vertical
            for l in range(barcos[b]):
                for di, dj in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1)
    , (1, 0), (1, 1)]:
                    ni, nj = i + l + di, j + dj
                    if 0 <= ni < n and 0 <= nj < m:
                        problema += (
                            lpSum(x[ni, nj, b2, o2] for (ni2, nj2, b2, o2) in x if
    ni2 == ni and nj2 == nj and b2 != b) <= 1 - x[i, j, b, o]
                        )


    # Resolver el modelo
    problema.solve(PULP_CBC_CMD())
```

De esta forma, se crea el modelo y se resuelve usando el solver de PULP, asignando las posiciones iniciales de los barcos de manera que se minimize la demanda incumplida total.

5.3. Resultados

Para mostrar la eficiencia del algoritmo de Programación Lineal Entera planteado, se ejecutará sobre el mismo set de datos usado para el algoritmo de Backtracking, y se compararán las mediciones de tiempos obtenidos.

5.3.1. Mediciones

Se muestran, en segundos, las mediciones temporales de ejecución de los algoritmos de Backtracking y de Programación Lineal Entera, ajustados a 5 cifras significativas. Se limitó hasta el set n°8 debido a la complejidad del algoritmo.

N° Set	1	2	3	4	5	6	7	8
Tiempo BT	0.0002	0.0002	0.0023	0.2467	0.0628	0.0399	0.4198	13.610
Tiempo PL	0.02	0.05	3.79	2273.5	1.96	41.26	125.28	> 5000

Cuadro 2: Tabla comparativa de tiempos de ejecución para PL y BT.

5.4. Análisis de los resultados

Se ve claramente que el algoritmo de programación lineal entera es significativamente mas lento que el de backtracking. Para sets de datos de pequeño y mediano tamaño (salvo el set 4) el algoritmo entrega la solución óptima en tiempos razonables, pero para sets grandes el algoritmo tarda un tiempo que hace que no valga la pena elegir este algoritmo por sobre el de backtracking.

6. Algoritmo de Aproximación

En la siguiente sección se analizara el algoritmo de aproximación planteado por John Jellicoe para resolver el problema de la Batalla Naval.

6.1. Algoritmo propuesto

El algoritmo propuesto por el almirante es el siguiente:

'Ir a fila/columna de mayor demanda, y ubicar el barco de mayor longitud en dicha fila/columna en algún lugar válido. Si el barco de mayor longitud es más largo que dicha demanda, simplemente saltearlo y seguir con el siguiente. Volver a aplicar hasta que no queden más barcos o no haya más demandas a cumplir.'

Para eso, planteamos el siguiente código:

```
def batalla_naval_individual_aprox(tablero, demanda_filas, demanda_columnas, barcos):

    indice_barco = 1
    flag_marcado = True
    n = len(tablero)
    m = len(tablero[0])

    # Mientras haya demanda por cumplir y barcos por ubicar
    while queda_demanda_por_cumplir(demanda_filas, demanda_columnas) and len(barcos) > 0:
        if not flag_marcado:
            break

        flag_marcado = False

        # Obtengo el indice de la fila/columna con mayor demanda y su tipo (fila/columna)
        indice_max_demanda, tipo = obtener_max_demanda(demanda_filas, demanda_columnas)

        for barco in barcos:

            # Obtengo una posicion valida para ubicar el barco de mayor longitud
            posicion = posicion_valida(tablero, barco, n, m, demanda_filas, demanda_columnas, indice_max_demanda, tipo)
            if posicion is None:
                continue

            if tipo == "fila":
                if demanda_filas[indice_max_demanda] >= barco:
                    marcar_barco_y_actualizar_demandas(tablero, barco, posicion, demanda_filas, demanda_columnas)
                    barcos.remove(barco)
                    indice_barco += 1
                    flag_marcado = True
                    break

            else:
                if barco <= len(tablero) and demanda_columnas[indice_max_demanda] >= barco:
                    marcar_barco_y_actualizar_demandas(tablero, barco, posicion, demanda_filas, demanda_columnas)
                    barcos.remove(barco)
                    indice_barco += 1
                    flag_marcado = True
                    break

            if flag_marcado:
                break

    return tablero, sum(demanda_filas) + sum(demanda_columnas)
```

Los barcos fueron ordenados de mayor a menor antes del llamado al algoritmo.

6.2. Análisis de la complejidad

A continuación analizaremos los fragmentos del algoritmo planteado y su complejidad, para poder analizar la complejidad de la totalidad del algoritmo de aproximación.

6.2.1. Preparación de la solución

Antes de llamar al algoritmo, se realizan algunas operaciones como:

- Construir el tablero vacio: $O(nxm)$
- Ordenar los barcos de mayor a menor: $O(klog(k))$, siendo k la cantidad de barcos.

6.2.2. Lógica principal

La función principal del algoritmo itera mientras haya demandas o barcos $O(k)$, y en cada iteracion se realizan las siguientes operaciones:

- Obtener la máxima demanda, con una complejidad de $O(n + m)$
- Por cada barco (k barcos):
  - Analizar si hay una posición válida $O(max(n, m) * b)$
  - Marcar el barco en el tablero y actualizar las demandas $O(b)$

Por lo que la complejidad de la función principal es $O(k * [n + m + k * max(n, m) * b])$

6.2.3. Complejidad total

Por lo tanto, la complejidad total del algoritmo de aproximación, teniendo en cuenta tanto la preparación de la solución como el bucle principal es:

$O(n * m + k * log(k) + k * [n + m + k * max(n, m) * b)$

Comparándolo con las soluciones por Backtracking o Programación Lineal Entera, que ambos tienen complejidad exponencial, parece reducir significativamente los tiempos de ejecución, ya que como peor caso no parece superar la complejidad cuadrática.

6.3. Mediciones

Los resultados de los tiempos de ejecución obtenidos utilizando el algoritmo de aproximación se muestran a continuación, utilizando el mismo set de datos que para el algoritmo de backtracking:

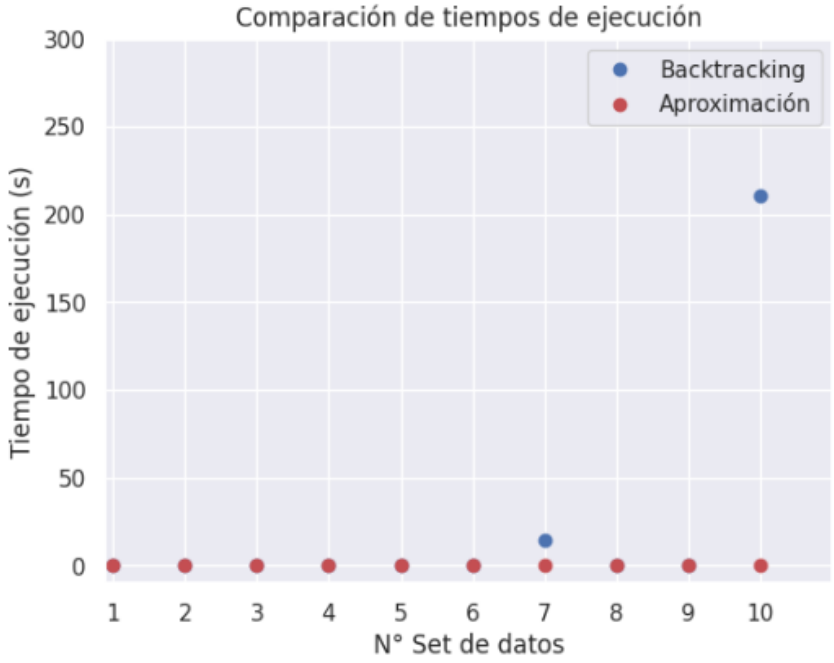

Figura 3: Gráfico comparativo de los tiempos de ejecución de Backtracking y Aproximación

Se ve claramente como el algoritmo de aproximación funciona de manera casi instantánea para todos los sets de datos, mientras que el algoritmo de backtracking sufre cuando se le pasan sets de datos de gran tamaño.

6.4. Análisis de la aproximación

6.4.1. Sets de datos brindados por la cátedra

A continuación se realizará una evaluación de qué tan buena es la aproximación realizada, comparando la solución aproximada con la solución óptima obtenida de los algoritmos de backtracking o programación lineal, utilizando los casos de pruebas brindados por la cátedra.

Para eso, definimos:

- I como una instancia cualquiera del problema de la Batalla Naval.
- $z(I)$ como la solución óptima para la instancia I.
- $A(I)$ como la solución aproximada para la instancia I.
- $r(A) \leq \frac{A(I)}{z(I)}$ una cota del peor caso que puede resultar de aplicar el algoritmo. Esto nos dará una idea de qué tan bueno es.

Podemos ver que la cota máxima de error es de 0.5643, es decir, que usando el algoritmo aproximado podemos obtener como peor caso, casi la mitad de la demanda cumplida óptima.

Caso de Prueba (I)	$z(I)$	$A(I)$	$r(A)$
3_3_2.txt	4	4	1
5_5_6.txt	12	12	1
8_7_10.txt	26	26	1
10_3_3.txt	6	6	1
10_10_10.txt	40	32	0.8
12_12_21.txt	46	40	0.8695
15_10_15.txt	40	38	0.95
20_20_20.txt	104	104	1
20_25_30.txt	172	152	0.8837
30_25_25.txt	202	114	0.5643

Cuadro 3: Tabla comparativa entre solución óptima y solución aproximada con casos de prueba.

6.4.2. Sets de datos generados aleatoriamente

Utilizando la función mostrada para el ejercicio de Backtracking, generaremos 10 sets de datos como se muestran a continuación, y se compararán los resultados para los algoritmos de backtracking y de aproximación:

N° Set	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Número de Barcos	2	4	6	8	10	12	14	16	18	20
Tamaño del Tablero	3x3	6x6	9x9	12x12	15x15	18x18	21x21	24x24	27x27	30x30

Cuadro 4: Sets de datos generados.

Set de datos (I)	$z(I)$	$A(I)$	$r(A)$
1	2	2	1
2	14	10	0.7142
3	30	30	1
4	76	24	0.3157
5	30	30	1
6	56	56	1
7	160	102	0.6375
8	226	204	0.9026
9	208	208	1
10	234	234	1

Cuadro 5: Tabla comparativa entre solución óptima y solución aproximada con sets de datos.

En este caso, podemos ver que la cota máxima de error es de 0.3157, es decir, que usando el algoritmo aproximado podemos obtener como peor caso, menos de la mitad de la demanda cumplida óptima.

6.5. Conclusión

Dado la cota calculada podemos decir que el algoritmo de aproximación planteado por John Jellicoe para resolver el problema de la Batalla Naval no es muy bueno. Vimos que en los sets generados aleatoriamente la relación entre la solución óptima y la aproximada, como peor caso, difiere en un factor de aproximadamente 3/10.

7. Conclusiones

En este último trabajo práctico se analizaron diferentes algoritmos para resolver el problema de la Batalla Naval, y se demostró que es un problema NP-Completo.

A continuación, haremos un análisis mas detallado de los resultados observados

7.1. Características de los algoritmos

Para resolver el problema de la Batalla Naval se utilizaron 3 algoritmos, cada uno con sus características propias y sus ventajas y desventajas:

- **Backtracking**: Devuelve siempre la solución óptima explorando todas las posibles configuraciones de barcos en el tablero brindado, optimizando la búsqueda con podas. Es un algoritmo que funciona muy rápido para sets de datos de pequeño y mediano tamaño, pero debido a su complejidad exponencial sufre con sets de datos de gran tamaño.
- **Programación Lineal Entera**: También devuelve siempre la solución óptima, modelando al problema como un sistema de ecuaciones lineales. Es un algoritmo que funciona rápido para sets de datos de pequeño tamaño, pero con algunas determinadas configuraciones, y sets de tamaño grande se hace inutilizable por el tiempo de ejecución. También esto se debe a su naturaleza exponencial.
- **Algoritmo de Aproximación**: Este algoritmo no devuelve la solución óptima y funciona con una rapidez significativamente mejor a las demás opciones. Dadas las mediciones, concluímos que el algoritmo planteado no es una buena aproximación, pero si se quiere tener una respuesta rápida para sets de datos inmanejables por los demas algoritmos, es una buena opción.

7.2. Observaciones

Más allá de el tamaño de los sets de datos utilizados en cada algoritmo, hemos notado que hay ciertas configuraciones de barcos y demandas que hacen que los algoritmos exactos (backtracking y progrmación lineal) funcionen con mayor lentitud. Esto se puede ver en las mediciones en el set n° 8 y 9 del algoritmo de backtracking, o en el set n° 4 de programación lineal entera.

Creemos que se puede deber a que ciertas configuraciones no encuentrar una solución lo suficientemente buena de manera rápida, por lo que tardan mucho en encontrar el óptimo.

7.3. Conclusión Final

Hemos aboradado el problema de la Batalla Naval desde distintas perspectivas, pudiendo profundizar en las características de cada enfoque, como sus ventajas y limitaciones. De esta forma, finalizamos el trabajo con un mejor entendimiento de qué analizar a la hora de tomar una decisión sobre que tipo de algoritmo utilizar.